Chapter V

Building Signature-Trees on Path Signatures in Document Databases

Yangjun Chen
University of Winnipeg, Canada

Gerald Huck
IPSI Institute, Germany

ABSTRACT

Java is a prevailing implementation platform for XML-based systems. Several high-quality in-memory implementations for the standardized XML-DOM API are available. However, persistency support has not been addressed. In this chapter, we discuss this problem and introduce PDOM (persistent DOM) to accommodate documents as permanent object sets. In addition, we propose a new indexing technique: path signatures to speed up the evaluation of path-oriented queries against document object sets, which is further enhanced by combining the technique of signature-trees with it to expedite scanning of signatures stored in a physical file.

INTRODUCTION

With the rapid advance of the Internet, management of structured documents such as XML documents has become more and more important (Suciu & Vossen, 2000; World Wide Web, 1998a; Marchiori, 1998). As a subset of SGML, XML is recommended by the W3C (World Wide Web Consortium) as a document description metalanguage to

exchange and manipulate data and documents on the WWW. It has been used to code various types of data in a wide range of application domains, including a Chemical Markup Language for exchanging data about molecules and the Open Financial Exchange for swapping financial data between banks, and between banks and customers (Bosak, 1997). Also, a growing number of legacy systems are adapted to output data in the form of XML documents.

In this chapter, we introduce a storage method for documents called *PDOM* (persistent DOM), implemented as a lightweight, transparent persistency memory layer, which does not require the burdensome design of a fixed schema. In addition, we propose a new indexing technique: *path signatures* to speed up the evaluation of path-oriented queries against document object sets, which are organized into a tree structure called *a signature-tree*. In this way, the scanning of a signature file is reduced to a binary tree search, which can be performed efficiently. To show the advantage of our method, the time complexity of searching a signature-tree is analyzed and the permanent storage of signature-trees is discussed in great detail.

BACKGROUND

The Document Object Model (DOM) is a platform- and language-neutral interface for XML. It provides a standard set of objects for representing XML data: a standard model of how these objects can be combined and a standard interface for accessing and manipulating them (Pixley 2000). There are half a dozen DOM implementations available for Java from several vendors such as IBM, Sun Microsystems and Oracle, but all these implementations are designed to work in main memory only. In recent years, efforts have been made to find an effective way to generate XML structures that are able to describe XML semantics in underlying relational databases (Chen & Huck, 2001; Florescu & Kossmann, 1999; Shanmugasundaram et al., 1999; Shanmugasundaram & Shekita, 2000; Yosjikawa et al., 2001). However, due to the substantial difference between the nested element structures of XML and the flat relational data, much redundancy is introduced, i.e., the XML data is either flattened into tuples containing many redundant elements, or has many disconnected elements. Therefore, it is significant to explore a way to accommodate XML documents, which is different from the relational theory. In addition, a variety of XML query languages have been proposed to provide a clue to manipulate XML documents (Abiteboul et al., 1996; Chamberlin et al., 2001; Christophides et al., 2000; Deutsch et al., 1989; Robie et al., 1998; Robie, Chamberlin & Florescu, 2000). Although the languages differ according to expressiveness, underlying formalism and data model, they share a common feature: *path-oriented queries*. Thus, finding efficient methods to do path matching is very important to evaluation of queries against huge volumes of XML documents.

SYSTEM ARCHITECTURE

The system architecture can be pictorially depicted as shown in Figure 1, which consists of three layers: persistent object manager, standard DOM API and specific PDOM API, and application support.

1. *Persistent Object Manager* — The PDOM mediates between in-memory DOM object hierarchies and their physical representation in binary random access files. The central component is the persistent object manager. It controls the life cycle of objects, serializes multi-threaded method invocations and synchronizes objects with their file representation. In addition, it contains two sub-components: a cache to improve performance and a commit control to mark recovery points in case of system crashes. These two components can be controlled by users through tuning parameters.
2. *Standard DOM API and Specific PDOM API* — The standard DOM API methods for object hierarchy manipulation are transparently mapped to physical file operations (read, write and update). The system aims at hiding the storage layer from an application programmer's view to the greatest possible extent. Thus, for most applications, it is sufficient to use only standard DOM methods. The only exception is document creation, which is deliberately left application-specific by the W3C DOM standard. The specific PDOM API allows an application to be aware of the PDOM to tune system parameters for the persistent object manager as well as its subsystems: cache and commit control. The specific API is mainly for the fine-grained control of the PDOM, not intended for the casual programmers. Rather, it is the place to experiment with ideas and proof concepts.
3. *Application Support* — This layer is composed of a set of functions which can be called by an application to read, write, update or retrieve a document. In addition, for a programmer with deep knowledge on PDOM, some functions are available to create a document, to commit an update operation and to compact a PDOM file, in which documents are stored as object hierarchies.

In the database (or PDOM pool), the DOM object hierarchies are stored as binary files while the index structures/path signatures are organized as a pat-tree.

Figure 1. Logical Architecture of the System

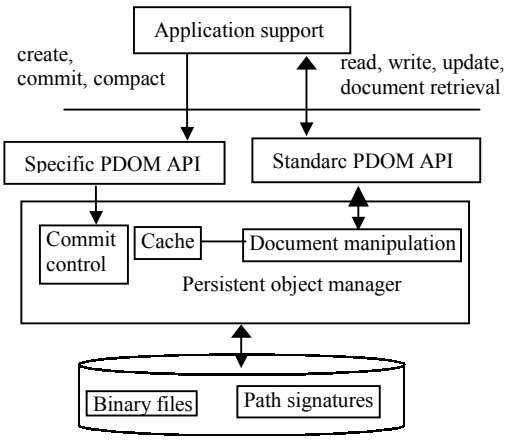
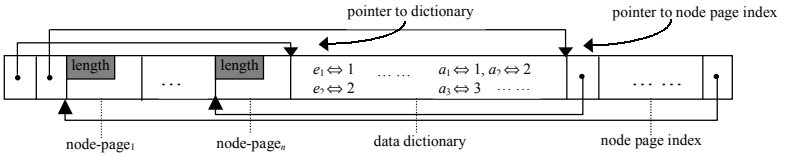


Figure 2. Binary File for Documents



STORAGE OF DOCUMENTS AS BINARY FILES

The format of the PDOM binary files used to accommodate the object hierarchies is depicted in Figure 2.

It is organized in node pages, each containing 128 serialized DOM objects. In PDOM, each object (node) corresponds to a document identifier, an element name, an element value, a "Comment" or a "Processing Instruction." The attributes of an element are stored with the corresponding element name. These object (node) types are equivalent to the node types in XSL (World Wide Web Consortium, 1998b) data model. Thus, a page does not have a fixed length in bytes, but a fixed number of objects it holds. At the beginning of the file, there are two pointers. The first points to a dictionary containing two mappings, by which each element name e_i and attribute a_j are mapped to a different number, respectively; the numerical values are used for compact storage. The second points to the node page index (NPI). The NPI holds an array of pointers to the start of each node page.

Each object is serialized as follows:

1. A *type flag* indicating the DOM-type: document identifier, element name, element value, "Comment" or "Processing Instruction."
2. The *object content* may be an integer representing an element name, a PCDATA (more or less comparable to a string) or a string (WTF-8 encoded) representing a "Comment" or a "Processing Instruction."
3. A *parent-element identifier* (if available).
4. A *set of attribute-value pairs*, where each attribute name is represented by an integer, which can be used to find the corresponding attribute name in the associated data dictionary. The attribute value is a WTF-8 encoded string.
5. The *number of sub-elements of an element and its sub-element identifiers.*

This serialization approach is self-describing, i.e., depending on the type flag, the serialization structure and the length of the remaining segments can be determined. The mapping between object identifiers in memory (OID) and their physical file location is given by the following equation:

$$OID = PI * 128 + i,$$

where *PI* is the index of the containing node page in the NPI and *i* is the object index within that page. Obviously, this address does not refer directly to any byte offset in the file

Figure 3. A Simple Document and its Storage

(a)

```
<letter filecode="9302">
    <date>January 27, 1993</date>
    <greeting>&salute; Jean Luc,</greeting>
    <body>
        <para>How are you doing?</para>
        <para>Isn't it
            <emph>about time</emph>
            you visit?
        </para>
    </body>
    <closing>See you soon,</closing>
    <sig>Genise</sig>
</letter>
```

(b)

byte number		
0:	500	pointer to the data dictionary
4:	565	pointer to the node page index
8:	0	first page number
9:	0	node type "document"
10:	1	number of children
11:	1	integer representing the child's id
12:	2	node type "element name"
13:	0	integer representing "letter"
14:	0	parent ID of this node
15:	1	number of attributes
16:	0	integer representing "filecode"
17:	"9302"	attribute value
22:	5	number of children
23:	2	ID of a child ("date" element)
...	...	...
		the following is the data dictionary
500:	7	number of element names
501:	letter	an element name "letter"
508:	date	an element name "date"
...	...	...
557	1	number of attribute names
...	...	...
565	8	

or page (which may change over time). Because of this, it can be used as unique, immutable object identifier within a single document. In the case of multiple documents, we associate each OID with a docID, to which it belongs. Example 1 helps for illustration.

Example 1

In Figure 3(a), we show a simple XML document. It will be stored in a binary file as shown in Figure 3(b).

From Figure 3(b), we can see that the first four bytes are used to store a pointer to the dictionary, in which an element name or an attribute name is mapped to an integer. (For example, the element name "letter" is mapped to "0," "date" is mapped to "1" and so on.) The second four bytes are a pointer to the node page index, which contains only one entry (four bytes) for this example, pointing to the beginning of the unique node page stored in this file. In this node page, each object (node) begins at a byte which shows the object type. In our implementation, five object types are considered. They are "document," "text" (used for an element value), "3," "4," respectively. The physical identifier of an object is implicitly implemented as the sequence number of the object appearing within a node page. For example, the physical identifier of the object with the type "document" is "0," the physical identifier of the object for "letter" is "1" and so on. The logic object identifier is calculated using the above simple equation when a node page is loaded into the main memory. Finally, we pay attention to the data dictionary structure. In the first line of the data dictionary, the number of the element names is stored, followed by the sequence of element names. Then, each element name is considered to be mapped implicitly to its sequence number in which it appears. The same method applies to the mapping for attribute names.

Beside the binary files for storing documents, another main data structure of the PDOM is the file for path signatures used to optimize the query evaluation. To speed up the scanning of the signatures, we organize them into a pat-tree, which reduces the time

complexity by an order of magnitude or more. We discuss this technique in the next section in detail.

PATH-ORIENTED LANGUAGE AND PATH SIGNATURES

Now we discuss our indexing technique. To this end, we first outline the path-oriented query language, which is necessary for the subsequent discussion. Then, the concept of path signatures will be described, and we will discuss the combination of path signatures and pat-trees, as well as the corresponding algorithm implementation in great detail.

Path-Oriented Language

Several path-oriented languages such as XQL (Robie et al., 1998) and XML-QL (Deutsch et al., 1998) have been proposed to manipulate tree-like structures as well as attributes and cross-references of XML documents. XQL is a natural extension to the XSL pattern syntax, providing a concise, understandable notation for pointing to specific elements and for searching nodes with particular characteristics. On the other hand, XML-QL has operations specific to data manipulation such as joins and supports transformations of XML data. XML-QL offers tree-browsing and tree-transformation operators to extract parts of documents to build new documents. XQL separates transformation operation from the query language. To make a transformation, an XQL query is performed first, then the results of the XQL query are fed into XSL (World Wide Web Consortium, 1998b) to conduct transformation.

An XQL query is represented by a line command which connects element types using path operators (‘/’ or ‘//’). ‘/’ is the child operator which selects from immediate child nodes. ‘//’ is the descendant operator which selects from arbitrary descendant nodes. In addition, the symbol ‘@’ precedes attribute names. By using these notations, all paths of tree representation can be expressed by element types, attributes, ‘/’ and ‘@’. Exactly, a simple path can be described by the following Backus-Naur Form:

<simplepath>::=<PathOP><SimplePathUnit>|<PathOp><SimplePathUnit>‘@’<AttName>
<PathOp> ::= ‘/’ | ‘//’
<SimplePathUnit>::=<ElementType> | <ElementType><PathOp><SimplePathUnit>

The following is a simple path-oriented query:

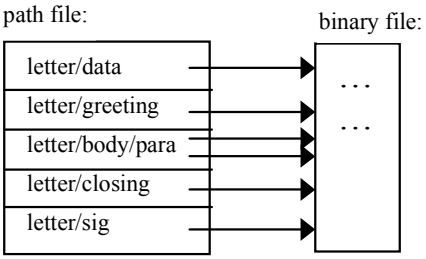
/letter//body [para $contains$’visit’],

where /letter//body is a path and [para $contains$’visited’] is a predicate, enquiring whether element “para” contains a word ‘visited.’

Signature and Path Signature

To speed up the evaluation of the path-oriented queries, we store all the different paths in a separate file and associate each path with a set of pointers to the positions of

Figure 4. Illustration for Path File



the binary file for the documents, where the element value can be reached along the path (see Figure 4 for illustration).

This method can be improved greatly by associating each path with a so-called path signature used to locate a path quickly. In addition, all the path signatures can be organized into a pat-tree, leading to a further improvement of performance.

Signature files are based on the inexact filter. They provide a quick test, which discards many of the nonqualifying values. But the qualifying values definitely pass the test, although some values which actually do not satisfy the search requirement may also pass it accidentally. Such values are called "false hits" or "false drops." The signature of a value is a hash-coded bit string of length k with m bit set to one, stored in the "signature file" (Faloutsos, 1985, 1992). The signature of an element containing some values is formed by superimposing the signatures of these values. The following figure depicts the signature generation and comparison process of an element containing three values, say "SGML," "database" and "information."

When a query arrives, the element signatures (stored in a signature file) are scanned and many nonqualifying elements are discarded. The rest are either checked (so that the "false drops" are discarded) or they are returned to the user as they are. Concretely, a query specifying certain values to be searched for will be transformed into a query signature s_q in the same way as for the elements stored in the database. The query signature is then compared to every element signature in the signature file. Three possible outcomes of the comparison are exemplified in Figure 3:

1. the element matches the query; that is, for every bit set to 1 in s_q, the corresponding bit in the element signature s is also set (i.e., $s \wedge s_q = s_q$) and the element really contains the query word;

Figure 5. Signature Generation and Comparison

text: ... SGML ... databases ... information ...

representative word signature:

SGML	010 000 100 110
database	100 010 010 100
information ∨	010 100 011 000
object signature (OS)	110 110 111 110

queries:	query signatures:	matchin results:
SGML	010 000 100 110	match with OS
XML	011 000 100 100	no match with OS
informatik	110 100 100 000	false drop

2. the element doesn't match the query (i.e., $s \wedge s_q \neq s_q$); and
3. the signature comparison indicates a match but the element in fact does not match the search criteria (false drop). In order to eliminate false drops, the elements must be examined after the element signature signifies a successful match.

The purpose of using a signature file is to screen out most of the nonqualifying elements. A signature failing to match the query signature guarantees that the corresponding element can be ignored. Therefore, unnecessary element accesses are prevented. Signature files have a much lower storage overhead and a simple file structure than inverted indexes.

The above filtering idea can be used to support the path-oriented queries by establishing path signatures in a similar way. First, we define the concept of tag trees.

Definition 1 (*tag trees*): Let d denote a document. A tag tree for d, denoted T_d, is a tree, where there is a node for each tag appearing in d and an edge ($node_a$, $node_b$) if $node_b$ represents a direct sub-element of $node_a$.

Based on the concept of tag trees, we can define path signatures as follows.

Definition 2 (*path signature*): Let $root \rightarrow n_1 \rightarrow ... \rightarrow n_m$ be a path in a tag tree. Let s_{root}, $s_i (i = 1, ..., m)$ be the signatures for *root* and $n_i (i = 1, ..., m)$, respectively.

The path signature of n_m is defined to be $Ps_m = s_{root} \vee s_1 \vee ... \vee s_m$.

Example 1

Consider the tree for the document shown in Figure 3(a). Removing all the leave nodes from it (a leaf always represents the text of an element), we will obtain the tag tree for the document shown in Figure 3(a). If the signatures assigned to 'letter,' 'body' and 'pare' are s_{letter} = 011 001 000 101, s_{body} = 001 000 101 110 and s_{para} = 010 001 011 100, respectively, then the path signature for 'para' is $Ps_{para} = s_{letter} \vee s_{body} \vee s_{para} = 011001111111$.

According to the concept of the path signatures, we can evaluate a path-oriented query as follows:

1. Assign each element name appearing in the path of the query a signature using the same hash function as for those stored in the path signature file.
2. Superimpose all these signatures to form a path signature of the query.
3. Scan the path signature file to find the matching signatures.
4. For each matching signature, check the associated path. If the path really matches, the corresponding page of the binary file will be accessed to check whether the query predicate is satisfied.

Compared to the path file, the path signature file has the following advantages:

i) If the paths (instead of the path signatures) are stored in a separate file, the path matching is more time-consuming than the path signatures. In the worst-case, O(n) time is required for a path matching, where n represents the length of the path (or the number of element names involved in a path). Assume that the average length of element names is w and each letter is stored as a bit string of length l. The time complexity of a path matching is then O($w \cdot l \cdot n$). But for a path signature matching,

only O(F) time is required, where F is the length of a path signature. In the terms of Christodoulakis and Faloutsos (1984), F is on the order of $O(m \cdot n / ln2)$, where m represents the number of 1s in a path signature (bit string). (Here, we regard each path as a "block" (Christodoulakis & Faloutsos, 1984), which is a set of words whose signatures will be superimposed together. Thus, the size of a block is the length of a path.) In general, $w \cdot l \geq m/ln2$. Therefore, some time can be saved using the path signatures instead of the paths themselves.

ii) We can organize all the path signatures into a pat-tree. In this way, the scanning of the path signatures can be expedited tremendously.

SIGNATURE-TREES ON PATH SIGNATURES

If a path signature file is large, the amount of time elapsed for scanning it becomes significant. Especially, the binary searching technique cannot be used to speed-up the searching of such a file since path signatures work only as an inexact filter. As a counter example, consider the following simple binary tree, which is constructed for a path signature file containing only three signatures (see Figure 6).

Assume that $s = 000010010100$ is a signature to be searched. Since $s_1 > s$, the search will go left to s_2. But s_2 does not match s. Then, the binary search will return a 'nil' to indicate that s cannot be found. However, in terms of the definition of the inexact matching, s_3 matches s. For this reason, we try another tree structure, the so-called *signature* index over path signatures, and change its search strategy in such a way that the behavior of signatures can be modeled. In the following, we first describe how to build a signature-tree. Then, we discuss how to establish an index for path signatures using signature-trees. Finally, we discuss how to search a signature-tree.

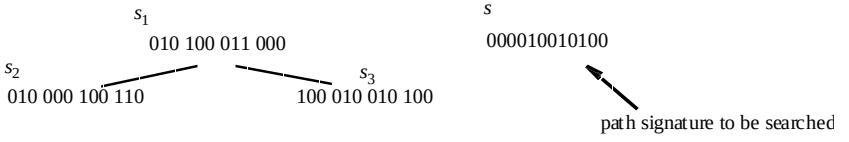
Definition of Signature-Trees

A signature-tree works for a signature file is just like a *trie* (Knuth, 1973; Morrison, 1968) for a text. But in a signature-tree, each path is a signature identifier which is not a continuous piece of bits, which is quite different from a trie in which each path corresponds to a continuous piece of bits.

Consider a signature s_i of length m. We denote it as $s_i = s_i[1]\, s_i[2] \ldots s_i[m]$, where each $s_i[j] \in \{0, 1\}$ $(j = 1, \ldots, m)$. We also use $s_i(j_1, \ldots, j_h)$ to denote a sequence of pairs w.r.t. s_i: $(j_1, s_i[j_1])(j_2, s_i[j_2]) \ldots (j_h, s_i[j_h])$, where $1 \leq j_{k\pm} m$ for $k \in \{1, \ldots, h\}$.

Definition 3 (*signature identifier*): Let $S = s_1.s_2 \ldots .s_n$ denote a signature file. Consider $s_i (1 \leq i \leq n)$. If there exists a sequence: $j_1, \ldots, j_h$ such that for any $k \neq i$ $(1 \leq k \leq n)$ we

Figure 6. A Counter Example



have $s_i(j_1, ..., j_h) \neq s_k(j_1, ..., j_h)$, then we say $s_i(j_1, ..., j_h)$ identifies the signature s_i or say $s_i(j_1, ..., j_h)$ is an identifier of s_i w.r.t. S.

For example, in Figure 6(a), $s_6(1, 7, 4, 5) = (1, 0)(7, 1)(4, 1)(5, 1)$ is an identifier of s_6 since for any $i \neq 6$ we have $s_i(1, 7, 4, 5) \neq s_6(1, 7, 4, 5)$. (For instance, $s_1(1, 7, 4, 5) = (1, 0)(7, 0)(4, 0)(5, 0) \neq s_6(1, 7, 4, 5)$, $s_2(1, 7, 4, 5) = (1, 1)(7, 0)(4, 0)(5, 1) \neq s_6(1, 7, 4, 5)$ and so on. Similarly, $s_1(1, 7) = (1, 0)(7, 0)$ is an identifier for s_1 since for any $i \neq 1$ we have $s_i(1, 7) \neq s_1(1, 7)$.)

In the following, we'll see that in a signature-tree each path corresponds to a signature identifier.

Definition 4 (*signature-tree*): A signature-tree for a signature file $S = s_1.s_2 ... .s_n$, where $s_i \neq s_j$ for $i \neq j$ and $|s_k| = m$ for $k = 1, ..., n$, is a binary tree T such that:

1. For each internal node of T, the left edge leaving it is always labeled with 0 and the right edge is always labeled with 1.
2. T has n leaves labeled 1, 2, ..., n, used as pointers to n different positions of s_1, s_2 ... and s_n in S.
3. Each internal node is associated with a number which tells how many bits to skip when searching.
4. Let $i_1, ..., i_h$ be the numbers associated with the nodes on a path from the root to a leaf labeled i (then, this leaf node is a pointer to the ith signature in S). Let $p_1, ..., p_h$ be the sequence of labels of edges on this path. Then, $(j_1, p_1) ... (j_h, p_h)$ makes up a signature identifier for $s_i, s_i(j_1, ..., j_h)$.

Example 2

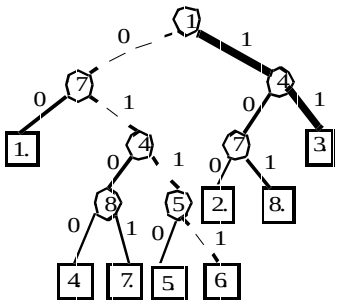
In Figure 7(b), we show a signature-tree for the signature file shown in Figure 6(a). In this signature-tree, each edge is labeled with 0 or 1 and each leaf node is a pointer to a signature in the signature file. In addition, each internal node is marked with an integer (which is not necessarily positive) used to calculate how many bits to skip when searching. Consider the path going through the nodes marked 1, 7 and 4. If this path is searched for locating some signature s, then three bits of s: $s[1]$, $s[7]$ and $s[4]$ will be checked at that moment. If $s[4] = 1$, the search will go to the right child of the node marked "4." This child node is marked with 5 and then the 5th bit of s: $s[5]$ will be checked.

Figure 7. A Path Signature FIle and its Signature Tree

(a)

s_1· 011 001 000 101
s_2· 111 011 001 111
s_3· 111 101 010 111
s_4· 011 001 101 111
s_5· 011 101 110 101
s_6· 011 111 110 101
s_7· 011 001 111 111
s_8· 111 011 111 111

(b)



See the path consisting of the dashed edges in Figure 7(b), which corresponds to the identifier of s_6: $s_6(1, 7, 4, 5) = (1, 0)(7, 1)(4, 1)(5, 1)$. Similarly, the identifier of s_3 is $s_3(1, 4) = (1, 1)(4, 1)$ (see the path consisting of thick edges).

In the next subsection, we discuss how to construct a signature-tree for a signature file.

Construction of Signature-Trees

Below we give an algorithm to construct a signature-tree for a signature file, which needs only O(N) time, where N represents the number of signatures in the signature file.

At the very beginning, the tree contains an initial node: a node containing a pointer to the first signature.

Then, we take the next signature to be inserted into the tree. Let s be the next signature we wish to enter. We traverse the tree from the root. Let v be the node encountered and assume that v is an internal node with $sk(v) = i$. Then, $s[i]$ will be checked. If $s[i] = 0$, we go left. Otherwise, we go right. If v is a leaf node, we compare s with the signature s_0 pointed by v. s cannot be the same as v since in S there is no signature which is identical to anyone else. But several bits of s can be determined, which agree with s_0. Assume that the first k bits of s agree with s_0; but s differs from s_0 in the $(k+1)$th position, where s has the digit b and s_0 has $1 - b$. We construct a new node u with $sk(u) = k + 1$ and replace v with u. (Note that v will not be removed. By "replace," we mean that the position of v in the tree is occupied by u. v will become one of u's children.) If $b = 1$, we make v and the pointer to s be the left and right children of u, respectively. If $b = 0$, we make v and the pointer to s be respectively the right and left children of u.

The following is the formal description of the algorithm.

```
Algorithm sig-tree-generation(file)
begin
    construct a root node r with sk(r) = 1; /*where r corresponds to the first signature
                                              s1 in the signature file*/
    for j = 2 to n do
        call insert(sj);
end

Procedure insert(s)
begin
    stack ← root;
    while stack not empty do
1       {v ← pop(stack);
2        if v is not a leaf then
3          {i ← sk(v);
4            if s[i] = 1 then {let a be the right child of v; push(stack, a);}
5            else {let a be the left child of v; push(stack, a);}
6          }
7        else     (*v is a leaf.*)
8            {compare s with the signature s0 pointed by p(v);
9            assume that the first k bit of s agree with s0;
```

```
10          but s differs from s_0 in the (k + 1)th position;
11          w ← v; replace v with a new node u with sk(u) = k + 1;
12          if s[k + 1] = 1 then
              make s and w be respectively the right and left
              children of u
13          else make s and w be the right and left children of u, respectively;}
14  }
end
```

In the procedure *insert*, *stack* is a stack structure used to control the tree traversal. We trace the above algorithm against the signature file shown in Figure 8.

In the following, we prove the correctness of the algorithm *sig-tree-generation*. To this end, it should be specified that each path from the root to a leaf node in a signature-tree corresponds to a signature identifier. We have the following proposition:

Proposition 1: Let T be a signature-tree for a signature file S. Let $P = v_1.e_1 \ldots v_g\text{-}1.e_g\text{-}1.v_g$ be a path in T from the root to a leaf node for some signature s in S, *i.e.*, $p(v_g) = s$. Denote $j_i = sk(v_i)$ $(i = 1, \ldots, g - 1)$. Then, $s(j_1, j_2, \ldots, j_{g-}1) = (j_1, b(e_1)) \ldots (j_{g-}1, b(e_g\text{-}1))$ constitutes an identifier for s.

Proof. Let $S = s_1.s_2 \ldots .s_n$ be a signature file and T a signature-tree for it. Let $P = v_1e_1 \ldots v_g\text{-}1e_g\text{-}1v_g$ be a path from the root to a leaf node for s_i in T. Assume that there exists another signature s_t such that $s_t(j_1, j_2, \ldots, j_{g-}1) = s_i(j_1, j_2, \ldots, j_{g-}1)$, where $j_i = sk(v_i)$ $(i = 1, \ldots, g - 1)$. Without loss of generality, assume that $t > i$. Then, at the moment when s_t is inserted into T, two new nodes v and v' will be inserted as shown in Figure 9(a) or (b) (see lines 10-15 of the procedure *insert*). Here, v' is a pointer to s_t and v is associated with a number indicating the position where $p(v_t)$ and $p(v')$ differs.

It shows that the path for s_i should be $v_1.e_1 \ldots v_g\text{-}1.e.ve'.v_g$ or $v_1.e_1 \ldots v_g\text{-}1.e.ve''.v_g$, which contradicts the assumption. Therefore, there is no other signature s_t with $s_t(j_1, j_2, \ldots, j_{n-}1) = (j_1, b(e_1)) \ldots (j_{n-}1, b(e_n\text{-}1))$. So $s_i(j_1, j_2, \ldots, j_{n-}1)$ is an identifier of s_i.

The analysis of the time complexity of the algorithm is relatively simple. From the procedure *insert*, we see that there is only one loop to insert all signatures of a signature

Figure 8. Sample Trace of Signature Tree Generation

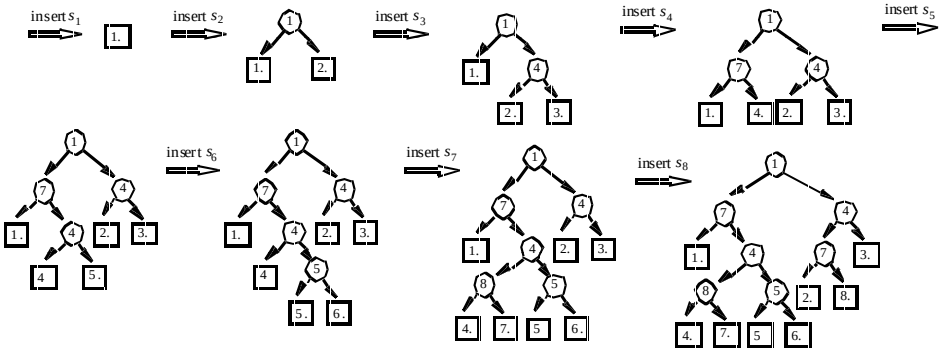


Figure 9. Inserting a Node 'v' into 'T'

file into a tree. At each step within the loop, only one path is searched, which needs at most O(m) time. (m represents the length of a signature.) Thus, we have the following proposition:

Proposition 2: The time complexity of the algorithm *sig-tree-generation* is bounded by O(N), where N represents the number of signatures in a signature file.

Proof. See the above analysis.

Searching of Signature-Trees

Now we discuss how to search a signature-tree to model the behavior of a signature file as a filter. Let s_q be a query signature. The i-th position of s_q is denoted as $s_q(i)$. During the traversal of a signature-tree, the inexact matching is defined as follows:

i) Let v be the node encountered and $s_q(i)$ be the position to be checked.
ii) If $s_q(i) = 1$, we move to the right child of v.
iii) If $s_q(i) = 0$, both the right and left child of v will be visited.

In fact, this definition corresponds to the signature matching criterion.

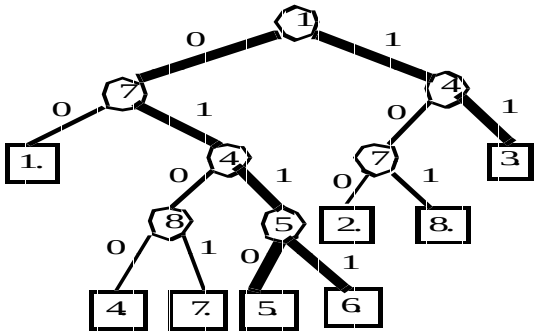
To implement this inexact matching strategy, we search the signature-tree in a depth-first manner and maintain a stack structure $stack_p$ to control the tree traversal.

Algorithm Signature-Tree-Search

input: a query signature s_q;
output: set of signatures which survive the checking;

1. $S \leftarrow$ Æ.
2. Push the root of the signature-tree into $stack_p$.
3. If $stack_p$ is not empty, $v \leftarrow$ pop($stack_p$); else return(S).
4. If v is not a leaf node, $i \leftarrow sk(v)$.
5. If $s_q(i) = 0$, push c_r and c_l into $stack_p$; (where c_r and c_l are v's right and left child, respectively) otherwise, push only c_r into $stack_p$.
6. Compare s_q with the signature pointed by $p(v)$. /*$p(v)$ - pointer to the block signature*/
   If s_q matches, $S \leftarrow S \cup \{p(v)\}$.
7. Go to (3).

Figure 10. Signature Tree Search



The following example helps to illustrate the main idea of the algorithm.

Example 3

Consider the signature file and the signature-tree shown in Figure 7(a) once again. Assume s_q = 000 100 100 000. Then, only part of the signature-tree (marked with thick edges in Figure 10) will be searched. On reaching a leaf node, the signature pointed by the leaf node will be checked against s_q. Obviously, this process is much more efficient than a sequential searching since only three signatures need to be checked while a signature file scanning will check eight signatures. For a balanced signature-tree, the height of the tree is bounded by $O(\log_2 N)$, where N is the number of the leaf nodes. Then, the cost of searching a balanced signature-tree will be $O(\lambda \cdot \log_2 N)$ on average, where λ represents the number of paths traversed, which is equal to the number of signatures checked. Let t represent the number of bits which are set in s_q and checked during the search. Then, $l = O(N/2^t)$. It is because each bit set to 1 will prohibit half of a subtree from being visited if it is checked during the search. Compared to the time complexity of the signature file scanning $O(N)$, it is a major benefit. We will discuss this issue in the next section in more detail.

Time Complexity

In this section, we compare the costs of signature file scanning and signature-tree searching. First, we show that a signature-tree is balanced on the average. Based on this, we analyze the cost of a signature-tree search.

Analysis of Signature-Trees

Let T_n be a family of signature-trees built from n signatures. Each signature is considered as a random bit string containing 0s and 1s. We assume that the probability of appearances of 0 and 1 in a string is equal to p and $q = 1 - p$, respectively. The occurrence of these two values in a bit string is independent of each other.

To study the average length of paths from the root to a leaf, we check the *external path length* L_n - the sum of the lengths of all paths from the root to all leaf nodes of a signature-tree in T_n. Note that in a signature-tree, the n signatures are split randomly into

the left subtree and the right subtree of the root. Let X denote the number of signatures in the left subtree. Then, for $X = k$, we have the following recurrence:

$$L_n = \begin{cases} n + L_k + L_{n-k}, & \text{for } k \neq 0, n \\ \text{undefined}, & \text{for } k = 0, k = n \end{cases}$$

where L_k and L_{n-k} represent the external path length in the left and right subtrees of the root, respectively. Note that a signature-tree is never degenerate (i.e., $k = 0$ or $k = n$). So one-way branching on internal nodes never happens. The above formula is a little bit different from the formula established for the external path length of a binary tree:

$$B_n = n + B_k + B_{n-k}, \quad \text{for all } k = 0, 1, 2, ..., n,$$

where B_k represents the sum of the lengths of all paths from the root to all leaf nodes of a binary tree having k leaf nodes.

According to Knuth (1973), the expectation of B_n is:

$$\boldsymbol{E}B_0 = \boldsymbol{E}B_1 = 0,$$

$$EB_n = \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} (B_k + B_{n-k}), \qquad n > 1.$$

When $p = q = 0.5$, we have:

$$EB_n = \sum_{k=2}^{n} (-1)^k \binom{n}{k} \frac{k}{1 - 2^{1-k}}.$$

For large n the following holds:

$$EB_n = n\log_2 n + n[\frac{\gamma}{L} + \frac{1}{2} + \delta_1(\log_2 n)] - \frac{1}{2} L + \delta_2(\log_2 n),$$

where $L = \log_e 2$, $\gamma = 0.577...$ is the *Euler* constant, $\delta_1(x)$ and $\delta_2(x)$ are two periodic functions with small amplitude and mean zero (see Knuth, 1973, for a detailed discussion).

In a similar way to Knuth (1973), we can obtain the following formulae:

$$\boldsymbol{E}L_0 = \boldsymbol{E}L_1 = 0,$$

$$\boldsymbol{E}L_n = n(1 - p^n - q^n) + \sum_{k=0}^{n} \binom{n}{k} p^k q^{n-k} (B_k + B_{n-k}), \qquad n > 1.$$

When $p = q = 0.5$, we have:

$$\boldsymbol{E}L_n = \sum_{k=2}^{n} (-1)^k \binom{n}{k} \frac{k2^{1-k}}{1-2^{1-k}} = \boldsymbol{E}B_n - n + \delta_{n,1},$$

where $\delta_{n,1}$ represents the *Kronecker delta function* (Riordan, 1968) which is 1 if $n = 1$, 0 otherwise.

From the above analysis, we can see that for large n we have the following:

$$\boldsymbol{E}L_n = O(n\log_2 n).$$

This shows that the average value of the external path length is asymptotically equal to $n\log_2 n$, which implies that a signature-tree is normally balanced.

Time for Searching a Signature-Tree

As shown in Example 4, using a balanced signature-tree, the cost of scanning a signature file can be reduced from $O(N)$ to $O(N/2^t)$, where t represents the number of some bits which are set in s_q and occasionally checked during the search. If $t = 1$, only half of the signatures will be checked. If $t = 2$, one-quarter of the signatures will be checked, and so on.

For a balanced signature-tree, the average height of the tree is $O(\log_2 N)$. During a search, if half of the s_q's bits checked are set to 1. Then, $t = O(\log_2 N)/2$. Accordingly, the cost of a signature file scanning can be reduced to $O(N/2^{(\log N)/2})$. If one-third of the s_q's bits checked are set to 1, the cost of a signature file scanning can be reduced to $O(N/2^{(\log N)/3})$.

Table 1 shows the calculation of this cost for different signature file sizes.

Figure 11 is the pictorial illustration of Table 1.

This shows that the searching of signature-trees outperforms the searching of signature files significantly.

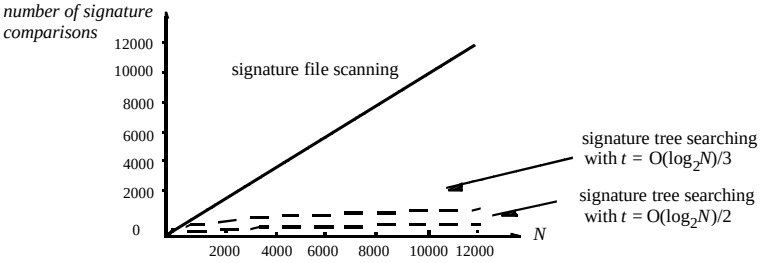
SIGNATURE-TREE MAINTANENCE

In this section, we address how to maintain a signature-tree. First, we discuss the case that a signature-tree can entirely fit in main memory. Then, we discuss the case that a signature-tree cannot entirely fit in main memory.

Table 1. Cost Calculation

N	2000	4000	6000	8000	10000	12000
$N/2^{(\log N)/2}$	44.68	63.36	77.76	89.44	100.00	109.56
$N/2^{(\log N)/3}$	159.36	251.92	330.10	399.98	463.90	524.13

Figure 11. Time Complexity of Signature File Scanning and Signature Tree Searching



Maintenance of Internal Signature-Trees

An internal signature-tree refers to a tree that can fit entirely in main memory. In this case, insertion and deletion of a signature into a tree can be done quite easily as discussed below.

When a signature *s* is added to a signature file, the corresponding signature-tree can be changed by simply running the algorithm *insert*() once with *s* as the input. When a signature is removed from the signature file, we need to reconstruct the corresponding signature-tree as follows:

i) Let *z*, *u*, *v* and *w* be the nodes as shown in Figure 12(a) and assume that the *v* is a pointer to the signature to be removed.

ii) Remove *u* and *v*. Set the left pointer of *z* to *w*. (If *u* is the right child of *z*, set the right pointer of *z* to *w*.)

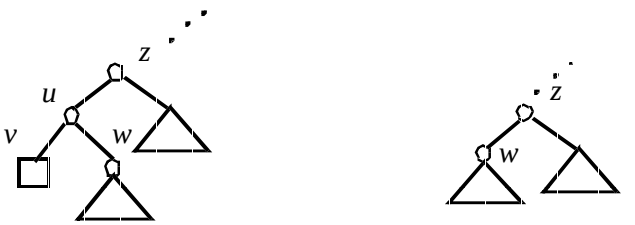
The resulting signature-tree is as shown in Figure 12(b).

From the above analysis, we see that the maintenance of an internal signature-tree is an easy task.

Maintenance of External Signature-Trees

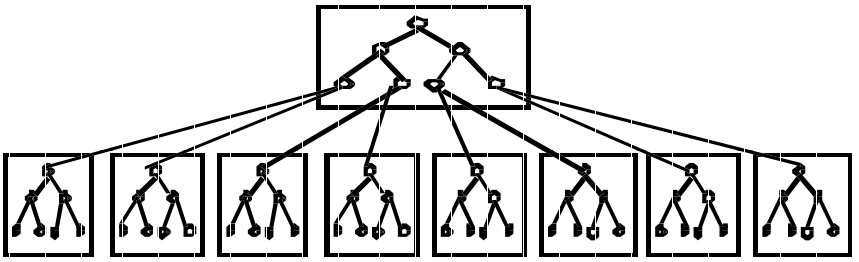
In a database, files are normally very large. Therefore, we have to consider the situation where a signature-tree cannot fit entirely in main memory. We call such a tree an external signature-tree (or an external structure for the signature-tree). In this case, a signature-tree is stored in a series of pages organized into a tree structure as shown in Figure 13, in which each node corresponds to a page containing a binary tree.

Figure 12. Illustration for Deletion of a Signature



(a) (b)

Figure 13. A Sample External Signature Tree



Formally, an external structure *ET* for a signature-tree *T* is defined as follows. (To avoid any confusion, we will, in the following, refer to the nodes in *ET* as the page nodes while the nodes in *T* as the binary nodes or simply the nodes.)

1. Each internal page node *n* of *ET* is of the form: $b_n(r_n, a_{n1}, ...,)$, where b_n represents a subtree of *T*, r_n is its root and a_{n1}, ..., are its leaf nodes. Each internal node *u* of b_n is of the form: $<v(u), l(u), r(u)>$, where $v(u)$, $l(u)$ and $r(u)$ are the value, left link and right link of *u*, respectively. Each leaf node of b_n is of the form: $<v(), lp(), rp()>$, where $v()$ represents the value of , and $lp()$ and $rp()$ are two pointers to two pages containing the left and right subtrees of, respectively.
2. Let *m* be a child page node of *n*. Then, *m* is of the form: $b_m(r_m, a_{m1}, ...,)$, where b_m represents a binary tree, r_m is its root and a_{m1}, ..., are its leaf nodes. If *m* is an internal page node, a_{m1}, ..., will have the same structure as a_{n1}, ..., described in (1). If *m* is a leaf node, each = $p(s)$, the position of some signature *s* in the signature file.
3. The size $|b|$ of the binary tree *b* (the number of nodes in *b*) within an internal page node of *ET* satisfies:

   $$|b| \le 2^k,$$

   where *k* is an integer.
4. The root page of *ET* contains at least a binary node and the left and right links associated with it.

If $2^k-1 \le |b| \le 2^k$ holds for each node in *ET*, it is said to be balanced; otherwise, it is unbalanced. However, according to the earlier analysis, an external signature-tree is normally balanced, i.e., $2^k-1 \le |b| \le 2^k$ holds for almost every page node in *ET*.

As with a B^+-tree, insertion and deletion of page nodes begin always from a leaf node. To maintain the tree balance, internal page nodes may split or merge during the process. In the following, we discuss these issues in great detail.

Insertion of Binary Nodes

Let *s* be a signature newly inserted into a signature file *S*. Accordingly, a node a_s will be inserted into the signature-tree *T* for *S* as a leaf node. In effect, it will be inserted into a leaf page node *m* of the external structure *ET* of *T*. It can be done by taking the binary tree within that page into main memory and then inserting the node into the tree. If for

the binary tree b in m we have $|b| > 2^k$, the following node-splitting will be conducted.

1. Let $b_m(r_m, a_{m1}, ...,)$ be the binary tree within m. Let r_{m1} and r_{m2} be the left and right child node of r_m, respectively. Assume that $b_{m1}(r_{m1}, a_{m1}, ...,)$ $(i_j < i_m)$ is the subtree rooted at r_{m1} and $b_{m2}(r_{m1}, , ...,)$ is rooted at r_{m2}. We allocate a new page m' and put $b_{m2}(r_{m1}, , ...,)$ into m'. Afterwards, promote r_m into the parent page node n of m and remove $b_{m2}(r_{m1}, , ...,)$ from m.
2. If the size of the binary tree within n becomes larger than 2^k, split n as above. The node-splitting repeats along the path bottom-up until no splitting is needed.

Deletion of Binary Nodes

When a node is removed from a signature-tree, it is always removed from the leaf level as discussed in the above subsection. *Let a* be a leaf node to be removed from a signature-tree *T*. In effect, it will be removed from a leaf page node m of the external structure *ET* for *T*. Let b be the binary tree within m. If the size of b becomes smaller than 2^k-1, we may merge it with its left or right sibling as follows.

1. Let m' be the left (right) sibling of m. Let $b_m(r_m, a_{m1}, ...,)$ and $b_{m'}(r_{m'}, a_{m'1}, ...,)$ be two binary trees in m and m', respectively. If the size of $b_{m'}$ is smaller than 2^k-1, move $b_{m'}$ into m and afterwards eliminate m'. Let n be the parent page node of m and r be the parent node of r_m and $r_{m'}$. Move r into m and afterwards remove r from n.
2. If the size of the binary tree within n becomes smaller than 2^k-1, merge it with its left or right sibling if possible. This process repeats along the path bottom-up until the root of *ET* is reached or no merging operation can be done.

Note that it is not possible to redistribute the binary trees of m and any of its left and right siblings due to the properties of a signature-tree, which may leave an external signature-tree unbalanced. According to our analysis, however, it is not a normal case.

Finally, we point out that for an application where the signature files are not frequently changed, the internal page nodes of an *ET* can be implemented as a heap structure. In this way, a lot of space can be saved.

CONCLUSION

In this chapter, a document management system is introduced. First, the system architecture and the document storage strategy have been discussed. Then, a new indexing technique, *path signature,* has been proposed to speed up the evaluation of the path-oriented queries. On the one hand, path signatures can be used as a filter to get away non-relevant elements. On the other hand, the technique of signature-trees can be utilized to establish index over them, which make us find relevant signatures quickly. As shown in the analysis of time complexity, high performance can be achieved using this technique.

REFERENCES

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. (1996). The Lorel Query Language for semi-structured data. *Journal of Digital Libraries,* 1(1).

Bosak, J. (1997, March). Java, and the future of the Web. Available online at: http://sunsite.unc.edu/pub/sun-info/standards/xml/why/xmlapps.html.

Chamberlin, D., Clark, J., Florescu, D., Robie, J., Simeon, J. & Stefanescu, M. (2001). *Xquery 1.0: An XML Query Language.* Technical Report, World Wide Web Consortium, Working Draft 07.

Chen, Y. & Huck, G. (2001). On the evaluation of path-oriented queries in document databases. *Lecture Notes in Computer Science,* 2113, 953-962.

Christodoulakis, S. & Faloutsos, C. (1984). Design consideration for a message file server. *IEEE Transactions on Software Engineering,* 10(2), 201-210.

Christophides, V., Cluet, S. & Simeon, J. (2000). On wrapping query languages and efficient XML integration. *Proceedings of the ACM SIGMOD Conference on Management of Data,* 141-152.

Deutsch, A., Fernandez, Florescu, D., Levy, A. & Suciu, D. (1988, August). *XML-QL: A Query Language for XML*. Available online at: http://www.w3.org/TR/NOTE-xml-ql/.

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, 17(1), 49-74.

Faloutsos, C. (1992). Signature files. In Frakes, W.B. & Baeza-Yates, R. (Eds.), *Information Retrieval: Data Structures & Algorithms*. Englewood Cliffs, NJ: Prentice Hall, 44-65.

Florescu, D. & Kossman, D. (1999). Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin,* 22(3).

Huck, G., Macherius, I. & Fankhauser, P. (1999). PDOM: Lightweight persistency support for the Document Object Model. *Proceedings of the OOPSLA'99 Workshop: Java and Databases: Persistence Options,* November.

Knuth, D.E. (1973). *The Art of Computer Programming: Sorting and Searching*. London: Addison-Wesley.

Marchiori, M. (1998). *The Query Languages Workshop (QL'98)*. Available online at: http://www.w3.org/TandS/QL/QL98.

Morrison, D.R. (1968). PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *Journal of Association for Computing Machinery*, 15(4), 514-534.

Pixley, T. (2000). *Document Object Model (DOM) Level 2 Events Specification Version 1.0*. W3C Recommendation.

Riordan, J. (1968). *Combinatorial Identities*. New York: John Wiley & Sons.

Robie, J., Chamberlin, D. & Florescu, D. (2000). Quilt: An XML query language for heterogeneous data sources. *Proceedings of the International Workshop on the Web and Databases*.

Robie, J., Lapp, J. & Schach, D. (1998). XML Query Language (XQL). *Proceedings of W3C QL'98—The Query Languages Workshop*.

Shanmugasundaram, J., Shekita, R., Carey, M.J., Lindsay, B.G., Pirahesh, H. & Reinwald, B. (2000). Efficiently publishing relational data as XML documents. *Proceedings of the International Conference on Very Large Data Bases (VLDB'00),* 65-76.

Shanmugasundaram, J., Tufte, K., Zhang. C., He,. D.J., DeWitt, J. & Naughton, J.F. (1999). Relational databases for querying XML documents: Limitations and opportunities. *Proceedings of the International Conference on Very Large Data Bases (VLDB'99),* 302-314.

Suciu, D. & Vossen, G. (2000). *Proceedings of the Third International Workshop on the Web and Databases (WebDB 2000),* LNCS. Springer-Verlag.

World Wide Web Consortium. (1998a, February). *Extensible Markup Language (XML) 1.0.* Available online at: http//www.w3.org/TR/ 1998/REC-xml/19980210.

World Wide Web Consortium. (1998b, December). *Extensible Style Language (XML) Working Draft.* Available online at: http/ /www.w3.org/TR/1998/WD-xsl-19981216.

World Wide Web Consortium. (1998c). *Document Object Model (DOM) Level 1.* Available online at: http://www.w3.org/TR/ REC-DOM-Level-1/.

Yoshikawa, M., Amagasa, T., Shimura, T. & Uemura, S. (2001). Xrel: A path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology,* 1(1).